# Real-time Operating Systems and Systems Programming

## Understanding Memory (Heap)
## Lecture 8

ARVUTITEADUSE INSTITUUT
DEPARTMENT OF COMPUTER SCIENCE

# Heap

- Section of memory for dynamic structures
- Bounded by brk pointer in kernel
- Function for allocation and deallocation:
  void *sbrk()
- Normally not used directly
  alloc(), malloc(), calloc(), free()
- Allocators divide heap into blocks

# Why dynamic allocation?

- Programs often know the amount of memory needed and sizes for data structures runtime
- RTOS note: you might still prefer static allocation for predictability

# Constraints for allocators

- Handling arbitrary request sequences
- Making immediate responses for requests
- Use only heap
- Block alignment must be kept
- Cannot modify allocated blocks

# Fragmentation problem

- Allocation and deallocation sequences can result in “holes”.
  - Internal fragmentation: the holes within memory blocks themselves
  - External fragmentation: happens when there would be enough free memory for a block, but a single block cannot hold it.

# Implementation

- Most naïve: just allocate, never reuse
- More clever:
  - Organize free blocks
  - Deal with placement of blocks
  - Splitting of blocks
  - Joining of blocks

# Organizing blocks

- Implicit free list
- Blocks have headers which include
  - Block size
  - Allocated/Free field
- Header size: 1 word
- Return the pointer to content, use header internally

# Header

- Due to alignment, the block sizes are multiple of 8
  - 3 lowest order bits are free!
  - Last bit used for free/allocated
- Terminating header with size 0
- “Contents” are located on double word alignment boundaries
- We have minimum block size

# Alignment trick

```
typedef long Align;

union Header {
    struct {
        union header *ptr;
        unsigned size;
    } s;
    Align x;
}
typedef union header Header;
```

# Where to place?

- When searching for a free block, one can have policies for placement:
  - First fit – end of list is often free; fragments
  - Next fit – spreads allocation; fragments worse
  - Best fit – good, but slower

# Should we split?

- Option to use entire block
- Or split
- If the fit is “good”, do not split

# How to get free memory?

- Ask for more (mmap() or sbrk() )
- Merge adjacent blocks upon freeing
  - Can also be done when needed

# Merging

- Merging next block is simple: just add
- How to find the previous block?
  - Boundary tags (block footer)
  - Block header has 2 free bits, use one to show that the previous block is free (then only free blocks have footers)

# Implementation details

- Initialize block list
- Decide policies
- Blocks may behave like data structures (linked or double linked lists)
- For faster allocation, keep free lists
- Segregation of free lists (see next)

# Simple Segregation

- For memory storage, a memory class will store blocks up to size X ( malloc({17-32}) → 32 )
- If new memory is needed, allocate a page
- Split it into equal blocks sized according to the storage class
- Do not merge blocks
- Link them into free list
- Problems: extreme fragmentation (sounds like a grenade)

# Segregated fit

- Allocator has an array of free lists, according to size classes
- Allocate according to class, first fit
- Split if needed
- If not found, search larger classes or ask more
- Thought to work well since GNU malloc() behaves like this

# Garbage collecting

- Each malloc() needs a free() to be called
- Mark & Block
  - Views the memory as a graph of directed nodes
  - Marks reachable nodes
  - Removes unreachable nodes